Improving software fault injection

van der Kouwe, E.

2016

document version
Publisher's PDF, also known as Version of record

Summary

Despite decades of advances in computer science, computer systems are still plagued by crashes and security lapses due to software faults caused by programmer mistakes. As a consequence, there is a great need for fault tolerance in software systems. To be able to compare such approaches and justify their cost, we need better ways to be able to determine how much they contribute to fault-tolerance in practical settings. To measure the fault tolerance of a software system, one must expose it to actual software faults. Software fault injection, also known as software mutation testing, is a suitable approach to achieve this purpose. By introducing deliberate software faults that are similar to those that could have been introduced by a human programmer, it becomes possible to perform a sufficiently large number of such experiments to be able to reach statistically significant conclusions. However, compared to other types of fault injection, software fault injection faces some unique challenges that make it harder to perform such experiments in a way that is representative of real-world faults. Because human mistakes cannot be predicted with purely theoretical models, they must be based on empirical analyses that investigate faults that humans have made in practice. A second difficulty is the fact that the system itself must be modified; it is not sufficient to alter the environment the system is running in. As a consequence, it is still hard to perform software fault injection efficiently in a methodologically sound way.

In this dissertation, we advance the state of the art in software fault injection to allow this technique to be used to measure fault tolerance in a way that is methodologically sound as well as efficient.

First, we raise the issue of fault load distortion. Our experiments demonstrate that fault activation correlates with factors that are important in defining the fault model, such as fault types and code locations with specific properties. We explain the implications of these results for workload selection and fault model specification.

By raising awareness of the issue of fault load distortion and providing guidelines to minimize its impact, we hope to increase the quality of the results from future software fault injection experiments.

Next, we present a widely applicable methodology to identify silent failures in fault injection experiments. Our approach consists of comparing externally visible behavior between a golden run using the original program and a faulty run where a fault has been injected in the program, while avoiding false positives due to non-determinism. We show that silent failures are common and consistent with field data, which suggests they they can be accurately emulated through fault injection. The main implication is that the fail-stop assumption does not hold, which means that fault-tolerance mechanisms depending on it need to be extended and re-evaluated using a method such as ours.

Then, we present an approach to compare the stability of operating systems. Rather then simply comparing the number of crashes when performing fault injection experiments, we argue that one cannot expect the operating system to function correctly if faults are injected. Instead, we use a pre-test and a post-test to determine whether running a workload that triggers the fault causes the system to become unstable afterwards. This allows for a meaningful comparison of the effectiveness of fault-tolerance mechanisms in operating systems. Another contribution of our methodology is a way to estimate how many fault injection experiments are needed to be able to draw statistical conclusions. Together, these elements provide a major step towards the systematic evaluation of fault-tolerance mechanisms in operating systems.

Finally, we present a methodology to run software fault injection experiments in such a way that source-level information is available to the fault injector while avoiding the slowdown caused by the need to recompile the source code for each experiment. The key is to create markers before code generation that can be recognized with very high accuracy in the resulting binary. Our evaluation compares this approach against alternatives and shows that it is the only one that can reach performance close to that of binary-level fault injection in all cases. This means high-quality fault injection experiments on large code bases can be run at lower cost.

Taken together, the contributions made in this dissertation allow software fault injection to be applied to evaluate fault-tolerance mechanisms in a way that is methodologically sound, low-cost even for large code bases such as operating systems and that provides state-of-the-art fault representativeness.